Prefix–suffix duplication

Jesús García López , Florin Manea , Victor Mitrana

Keywords:
Bio-inspired operations
Prefix–suffix duplication
Language theoretical properties
Prefix–suffix duplication distance
Algorithms on words

A B S T R A C T

We consider a bio-inspired formal operation on words called prefix–suffix duplication which consists in the duplication of a prefix or suffix of a given word. The class of languages defined by the iterated application of the prefix–suffix duplication to a word is considered. We show that such a language is context-free if and only if the initial word contains just one letter. Moreover, every language in this class is semilinear and belongs to **NL**. We propose a $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n^2)$ space recognition algorithm. Two algorithms are further proposed for computing the prefix–suffix duplication distance between two words, defined as the minimal number of prefix–suffix duplications applied to one of them in order to get the other one. The first algorithm runs in cubic time and uses quadratic space while the second one is more efficient, having $\mathcal{O}(n^2 \log n)$ time complexity, but needs $\mathcal{O}(n^2 \log n)$ space.

1. Introduction

One of the most frequent and less understood mutations among the genome rearrangements is the duplication of a segment of a chromosome [15]. In the process of duplication, a stretch of DNA is duplicated yielding two or more adjacent copies, called also tandem repeats. It is commonly asserted that approximately 5% of the genome is involved in duplications and the distribution of these tandem repeats varies widely along the chromosomes [20]. An interesting property of tandem repeats is to make possible a so-called "phylogenetic analysis" which might be useful in the investigation of the evolution of species by determining the most likely duplication history [22]. The detection of these tandem repeats and algorithms for tandem repeats reconstructing history have received a great deal of attention in bioinformatics [1,2,19].

However, a special type of duplications, known as telomeres, appear only at the ends of chromosomes. Generally, telomeres consist of tandem repeats of a small number of nucleotides, specified by the action of telomerase. They are considered to be protective DNA-protein complexes found at the end of eukaryotic chromosomes which stabilize the linear chromosomal DNA molecule [4,16]. The length of telomeric DNA is important for the chromosome stability: the loss of telomeric repeat sequences may result in chromosome fusion and lead to chromosome instability [12]. In [20] one states that it is a further challenge the sequencing of the 20% of the genome that is formed by repetitive heterochromatin which is implicated in the process of chromosome replication and maintenance.

Treating chromosomes and genomes as languages raises the possibility that the structural information contained in biological sequences can be generalized and investigated by formal language theory methods [18]. Thus, the interpretation of duplication as a formal operation on words has inspired several works in the area of Formal Languages opened by [6,21] and continued in a series of papers, see, e.g., [11] and the references therein. In the first part of this paper we follow a similar approach to that from [6,21,10]. We consider only duplications that may appear in the ends of the words only, called prefix–suffix duplications, similar to the case of telomeric DNA. In this context, we investigate the class of languages that can be defined by the iteratively application of the prefix–suffix duplication to a word and try to compare it to other well studied classes of languages. To this end, we show that the languages of this class have a rather complicated structure even if the initial word is rather simple; more precisely, they are already non-context-free as soon as the initial word contains at least two different letters. Consequently, one can derive a trivial algorithm deciding in linear time whether the prefix–suffix duplication language defined by a given word is context-free (or equivalently, in the case of unary languages, regular) just by counting the different letters that occur in the given word. Naturally, we also investigate how complicated these languages actually are, or, formally, try to derive upper bounds for the class of prefix–suffix duplication languages. We show that all the languages of this class have a linear Parikh image and belong to **NL**, hence are polynomially recognizable. Starting from this result, we focus on the computational complexity of solving problems related to such languages. First, we are interested in finding an efficient algorithm solving the membership problem for such languages. To this aim, in the second part of the paper, we propose a $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n^2)$ space recognition algorithm. Then, we consider the prefix–suffix duplication distance between two given words, defined as the minimal number of prefix–suffix duplications applied to one word in order to get the other, and develop efficient algorithmic solutions to compute it. We propose two algorithms: a cubic time one which uses a quadratic memory and a more efficient one, namely $\mathcal{O}(n^2 \log n)$ time complexity, but with some extra memory consumption, that is $\mathcal{O}(n^2 \log n)$ space complexity. It is worth mentioning that the efficiency of the algorithms we present follows from the application of a series of non-trivial combinatorics on words remarks as well as the usage of several data-structures, specific to stringology.

One should note that the investigation we pursue here is not aimed to tackle real biological solutions. In fact, its aim is to provide a better understanding of the structural properties of strings obtained by prefix–suffix duplication as well as specific tools for the manipulation of such strings. On the long run, such tools could provide the foundations on which applications working with real data are built.

2. Preliminaries

We assume the reader to be familiar with fundamental concepts from Formal Language Theory, such as the classes of the Chomsky hierarchy, finite automaton, generalized sequential machine (gsm), which can be found in many textbooks, e.g., the handbook [17]. The same assumption concerns fundamental concepts from Complexity Theory such as Turing machine, random access machine (RAM) with logarithmic word size and standard unit-cost operations, time and space complexity classes; see, for instance, [14].

We start by summarizing the notions used throughout this work. An *alphabet* is a finite and nonempty set of symbols. The cardinality of a finite set A is written $card(A)$. Any finite sequence of symbols from an alphabet V is called *word* over V. The set of all words over V is denoted by V^* and the empty word is denoted by ε; we further let $V^+ = V^* \setminus \{\varepsilon\}$. Given a word w over an alphabet V, we denote by $|w|$ its length, while $|w|_a$ denotes the number of occurrences of the letter a in w. Furthermore, $alph(w)$ denotes the minimal alphabet W such that $w \in W^*$, i.e. $alph(w) = \{a \in V \mid |w|_a \neq 0\}$. If $w = xyz$ for some $x, y, z \in V^*$, then x, y, z are called prefix, subword, suffix, respectively, of w. For a word w, $w[i..j]$ denotes the subword of w starting at position i and ending at position j, $1 \leqslant i \leqslant j \leqslant |w|$; by convention, $w[i..j] = \varepsilon$ if $i > j$. If $i = j$, then $w[i..j]$ is the i-th letter of w which is simply denoted by $w[i]$. A *period* of a word w over V is a positive integer p such that $w[i] = w[j]$ for all i and j with $i \equiv j \pmod p$. By $per(w)$ (called *the period of* w) we denote the smallest period of w.

We say that the pair ${}_w(i, p)$ is a *duplication* (*repetition*) in w starting at position i in w if $w[i..i+p-1] = w[i+p..i+2p-1]$. Analogously, the pair $(i, p)_w$ is a duplication in w ending at position i in w if $w[i-2p+1..i-p] = w[i-p+1..i]$. In both cases, p is called the length of the duplication.

Given a word x over an alphabet V, we consider the following duplication operations:

- *Prefix duplication*, namely $PD(x) = \{ux \mid x = uy \text{ for some } u \in V^+\}$. The *suffix duplication* is defined analogously, that is $SD(x) = \{xu \mid x = yu \text{ for some } u \in V^+\}$.
- *Prefix–suffix duplication*, namely $PSD(x) = PD(x) \cup SD(x)$.

The prefix–suffix duplication is naturally extended to languages L by $PSD(L) = \bigcup_{x \in L} PSD(x)$. We further define:

$$PSD^0(x) = \{x\},$$
$$PSD^{k+1}(x) = PSD^k(x) \cup PSD\big(PSD^k(x)\big), \quad \text{for any } k \geqslant 0,$$
$$PSD^*(x) = \bigcup_{k \geqslant 0} PSD^k(x).$$

We say that a language $L \subseteq V^*$ is a prefix–suffix duplication language if $L = PSD^*(x)$ for some $x \in V^*$. An arbitrary duplication language is defined analogously, see, e.g., [6], with the difference that duplications within the word are also permitted.

The prefix–suffix duplication distance between two words w and x is defined as follows:

$$\pi(w,x) = \begin{cases} \text{the minimum number } \ell \text{ such that } w \in PSD^{\ell}(x) \text{ or } x \in PSD^{\ell}(w), \\ \infty, \quad \text{if } w \notin PSD^*(x) \text{ and } x \notin PSD^*(w). \end{cases}$$

We stress from the very beginning that the function π, applied on pairs of words, is not a distance function in the strict mathematical sense, since it does not necessarily verify the triangle inequality. It can be rather seen as a similarity measure between strings, or, if we consider our biological motivation, a measure that tells us how many evolution steps are needed to transform a string into the other. However, we call it distance in order to make the exposure more cursive.

Note that if $\pi(x, w)$ is defined, then $0 \leqslant \pi(x, w) \leqslant ||x| - |w||$.

3. Prefix–suffix duplication languages

In this section we present some language theoretical properties of the class of prefix–suffix duplication languages. Before doing this we recall some results known for arbitrary duplication languages.

By combining the results from [3] and [7] (rediscovered in [6] and [21] for arbitrary duplication languages) we recall that

Theorem 1. *An arbitrary duplication language is regular if and only if it is a language over an alphabet with at most two symbols.*

An open problem asks weather or not there exist arbitrary duplication languages that are not context-free. Weather or not every arbitrary duplication language is recognizable in polynomial time is open as well. The fact that the Parikh image of an arbitrary duplication language is linear is immediate.

We now try to answer these questions for prefix–suffix duplication languages. Much differently from the situation for arbitrary duplication languages we have:

Theorem 2. *A prefix–suffix duplication language is context-free if and only if it is a language over the unary alphabet.*

Proof. The fact that the prefix–suffix duplication of a word over a unary alphabet is a regular (thus, context-free) language follows immediately from Theorem 1 as, in that case, the arbitrary duplication operation is equivalent to prefix–suffix duplication.

We first show that the prefix–suffix duplication language generated by ab is non-context-free. The idea will be later extended to every word containing at least two letters.

Claim. $PSD^*(ab) \cap ab^+ab^+ab^+ = \{ab^m ab^n ab^p \mid n, m, p \geqslant 1, m \leqslant \min(n, p) \text{ and } n \leqslant m + p\}$.

Proof of the Claim. We first show that every word $w = ab^m ab^n ab^p$, with $1 \leqslant m \leqslant \min(n, p)$ and $n \leqslant m + p$, belongs to $PSD^*(ab)$. Assume that $m \leqslant n \leqslant p$, more precisely $n = m + r$ and $p = n + s$ for some $r, s \geqslant 0$. The word w can be obtained from ab by prefix–suffix duplications as follows:

(i) $ab^m \in PSD^*(ab)$,

(ii) $ab^m ab^m \in PSD^*\left(ab^m\right)$,

(iii) $ab^m ab^{m+r} \in PSD^*\left(ab^m ab^m\right)$,

(iv) $ab^m ab^n ab^n \in PSD^*\left(ab^m ab^n\right)$,

(v) $ab^m ab^n ab^{n+s} \in PSD^*\left(ab^m ab^n ab^n\right)$.

Assume now that $m \leqslant p < n \leqslant m + p$, more precisely $n = m + p - r$ with $r < m$ and $p = m + s$; note that $0 \leqslant s < p - r$ holds. The word w can be obtained from ab by prefix–suffix duplications as follows:

(i) $ab^m \in PSD^*(ab)$,

(ii) $ab^m ab^m \in PSD^*\left(ab^m\right)$,

(iii) $ab^m ab^{m+s} \in PSD^*\left(ab^m ab^m\right)$,

(iv) $ab^m ab^{m+s+m-r} ab^{m+s} \in PSD^*\left(ab^m ab^{m+s}\right)$.

As $n = m + s + m - r$, we are done.

Conversely, we analyze how a word in $PSD^*(ab) \cap ab^+ab^+ab^+$ looks like. Assume that starting from ab after a number of prefix–suffix duplications which does not increase the number of adjacent occurrences of a we get ab^m for some $m \geqslant 1$. We now want to produce either an a after the rightmost b or a b before the leftmost a in ab^m. There are two possible situations: the next word is either ab^sab^m, with $1 \leqslant s \leqslant m$, or ab^mab^m. Note that the first word was obtained by prefix duplication while the second one was obtained by suffix duplication. We continue our discussion with the word ab^sab^m, with $1 \leqslant s \leqslant m$, which covers the both cases. The word ab^sab^m may be extended to ab^sab^n, with $s \leqslant m \leqslant n$, by successive suffix duplication steps; any prefix duplication applied to this word would produce a word with at least three occurrences of the letter a. To end our derivation, we now try to get a word in $ab^+ab^+ab^+$. This can be done in two ways: (i) by a prefix duplication that leads to $ab^jab^sab^n$ for some $1 \leqslant j \leqslant s$, or (ii) by a suffix duplication that leads to $ab^sab^{n+k}ab^n$ for some $0 \leqslant k \leqslant s$. In the first case, when the current word is $ab^jab^sab^n$, then $1 \leqslant j \leqslant s \leqslant n$ holds. If the current word is $ab^sab^{n+k}ab^n$, then $1 \leqslant s \leqslant \min(n+k, n)$ and $n+k \leqslant n+s$ hold as well. Any of these word can be extended, such that a word from our target set $ab^+ab^+ab^+$ is obtained, only by duplication of suffixes b^ℓ with $\ell \geqslant 1$; thus, only n can increase. Thus, no matter how many further prefix–suffix duplication steps are made, the obtained words remain in the set stated by the claim. This concludes the proof of the claim.

The language $\{ab^mab^nab^p \mid n, m, p \geqslant 1, m \leqslant \min(n, p) \text{ and } n \leqslant m+p\}$ is not context-free. Indeed, this language is rejected to be context-free by Ogden Lemma [13], as soon as we mark all occurrences of b before the second occurrence of a. Since $ab^+ab^+ab^+$ is a regular language and the intersection of a context-free language and a regular language is still context-free, we conclude that $PSD^*(ab)$ is not context-free.

We now extend this argument to prove that the prefix–suffix duplication language defined by a word containing at least two different letters is not context-free. Let $w \in a_1^+a_2^+ \cdots a_r^+$ be an arbitrary word over an alphabet with at least two letters such that each a_i is a letter of this alphabet. Assume that $a_i \neq a_{i+1}$ for all $1 \leqslant i \leqslant r-1$, hence w can be written as $w = a_1^{k_1}a_2^{k_2}\ldots a_r^{k_r}$, for some $r \geqslant 2$, $k_j \geqslant 1$ for all $1 \leqslant j \leqslant r$. We state that $PSD^*(w)$ is not context-free. The statement is true for $r = 2$ by a similar argument to that from above.

Now, let $r \geqslant 3$. We set

$$k = \max\{k_1, k_2, \ldots, k_r\} + 1.$$

We consider a gsm mapping g that successfully terminates its computation on words in $a_1^+a_2^+\ldots a_r^+a_{r-1}^+a_r^+a_{r-1}^+a_r^+$ only, and keeps from each such input word only its suffix in $a_{r-1}^+a_r^+a_{r-1}^+a_r^+a_{r-1}^+a_r^+$. We define the regular language

$$R = \left\{a_1^{k_1}a_2^{k_2}\ldots a_{r-1}^{k_{r-1}}a_r^m a_{r-1}^{k_{r-1}}a_r^n a_{r-1}^{k_{r-1}}a_r^p \;\middle|\; m, n, p \geqslant k\right\}.$$

We distinguish two cases.

Case 1: $a_{r-2} \neq a_r$. Note that all operations used in the previous reasoning showing that $PSD^*(ab)$ is not context-free are suffix duplications. An analogous argument leads to the following equation:

$$g\big(PSD^*(w) \cap R\big) = \left\{a_{r-1}^{k_{r-1}}a_r^m a_{r-1}^{k_{r-1}}a_r^n a_{r-1}^{k_{r-1}}a_r^p \;\middle|\; m, n, p \geqslant k,\ m \leqslant \min(n, p) \text{ and } n \leqslant m+p\right\}.$$

Case 2: $a_{r-2} = a_r$. By the choice of k, the equation above becomes:

$$g\big(PSD^*(w) \cap R\big) = \left\{a_{r-1}^{k_{r-1}}a_r^m a_{r-1}^{k_{r-1}}a_r^n a_{r-1}^{k_{r-1}}a_r^p \;\middle|\; m, n, p \geqslant k,\ m \leqslant \min(n+k_{r-2}, p+k_{r-2}) \text{ and } n \leqslant m+p\right\}.$$

Since the image of the gsm mapping in both cases is a non-context-free language, by the closure properties of the class of context-free languages, we conclude that $PSD^*(w)$ is not context-free. □

An immediate consequence of this result is:

Corollary 1. *It is algorithmically decidable in linear time whether or not a prefix–suffix duplication language is regular or context-free.*

Further, we present a series of upper bounds on the class of prefix–suffix duplication languages.

Theorem 3. *The Parikh image of every prefix–suffix duplication language is a linear set.*

Proof. Let x be a word of length n over an arbitrary alphabet. We claim that:

$$\Psi\left(PSD^*(x)\right) = \left\{\Psi(x) + \sum_{i=1}^{n} l_i p_i + \sum_{i=1}^{n} r_i s_i \;\middle|\; l_i, r_i \in \mathbb{N}\right\},$$

where Ψ is the Parikh mapping and

$$p_i = \Psi\big(x[1..i]\big), \qquad s_i = \Psi\big(x[i..n]\big), \quad 1 \leqslant i \leqslant n.$$

For simplicity we denote $X = \{\Psi(x) + \sum_{i=1}^{n} l_i p_i + \sum_{i=1}^{n} r_i s_i \mid l_i, r_i \in \mathbb{N}\}$.

Algorithm 1 Procedure for computing w. **Input:** x and t.

```
1: w := x
2: for i = n downto 1 do
3:     for j = 1 to l_i do
4:         w := PD_i(w);
5:     end for
6: end for
7: for i = 1 to n do
8:     for j = 1 to r_i do
9:         w := SD_i(w);
10:    end for
11: end for
```

Algorithm 2 Function $Member(w, i, j, x)$.

```
1: Member := false;
2: if w[i..j] = x then
3:     Member := true; HALT;
4: end if
5: Choose non-deterministically 1 ⩽ k ⩽ ⌊(j−i+1)/2⌋
6: Proceed non-deterministically with
7: if (w[i..i+k−1] = w[i+k..i+2k−1]) and Member(w, i+k, j, x) then
8:     Member := true; HALT;
9: end if
10: if w[j−2k+1..j−k] = w[j−k+1..j] and Member(w, i, j−k, x) then
11:    Member := true; HALT;
12: end if
```

Claim. *If $w \in PSD^*(x)$, then for any prefix or suffix y of w, $\Psi(y) \in X$.*

Proof of the Claim. The proof is based on the induction on the number of prefix–suffix duplication applied to x in order to get w, that is $w \in PSD^k(x)$. The induction basis is verified immediately as $PSD^0(x) = \{x\}$. Assume now that $w \in PSD^{k+1}(x)$; there exists $z \in PSD^k(x)$ such that $w \in PSD(z)$. Let y be an arbitrary prefix of w; the case when y is a suffix can be treated analogously. We distinguish two cases:

Case 1: w was obtained from z by prefix duplication. This means that $w = uuv$ such that $z = uv$.

If y is a prefix of u or equals u, then $\Psi(y) \in X$ by the induction hypothesis. If $y = u\alpha$, then $\Psi(y) = \Psi(u) + \Psi(\alpha)$. Consequently, $\Psi(y) \in X$ follows as both u and α are prefixes of z.

Case 2: w was obtained from z by suffix duplication. This means that $w = vuu$ and $z = vu$.

If y is a prefix of z, then we are done by the induction hypothesis. If $y = zu_1$ such that $u = u_1u_2$, then $\Psi(y) = \Psi(vu_1) + \Psi(u_2u_1)$. As vu_1 is a prefix of z and $\Psi(u_2u_1) = \Psi(u)$ we are done.

A direct consequence of this claim is that $PSD^*(x) \subseteq X$.

For the converse inclusion, we take $t \in X$,

$$t = \Psi(x) + l_1p_1 + l_2p_2 + \cdots + l_np_n + r_1s_1 + r_2s_2 + \cdots + r_ns_n,$$

for some natural numbers l_i, r_i, $1 \leqslant i \leqslant n$. We construct w such that $w \in PSD^*(x)$ and $\Psi(w) = t$. More precisely, Algorithm 1 outputs the word w. Here $PD_i(w)$ ($SD_i(w)$) denotes the prefix (suffix) duplication of w where the duplicated prefix (suffix) is of length i.

Now the proof is complete. □

Theorem 4. *Every prefix–suffix duplication language is in* **NL**.

Proof. The recursive boolean function $Member(w, i, j, x)$, computed non-deterministically in Algorithm 2, determines whether or not $w[i..j] \in PSD^*(x)$. This function can clearly be implemented on a nondeterministic (multi-tape) off-line Turing machine in $\mathcal{O}(\log n)$ space. □

A direct consequence of the last result is that every prefix–suffix duplication language can be recognized in polynomial time by Turing machines. Note, however, that the previous theorem has an emphasized computational complexity flavor, showing that a certain class of languages is included in a classically defined complexity class, namely **NL**. Hence, we decided to use in its proof the classical computational model, Turing machines. Accordingly, when determining the space complexity of Algorithm 2 we assumed that numbers less or equal to n are stored on $\mathcal{O}(\log n)$ bits.

However, as efficient algorithms and data structures are crucial tools in the area of computational biology, to which our investigation tries to contribute, we would prefer finding a more precise bound on the time complexity of the recognition of prefix–suffix duplication languages, on a computational model closer to practice. Therefore, we decided to choose for this

purpose as a new computational model the more realistic RAM model with logarithmic word size, where, compared to the Turing machines, we assume that for an input of length n each memory cell can store $\mathcal{O}(\log n)$ bits, or, in other words, that *the machine word size* is $\mathcal{O}(\log n)$. In the RAM model we assume that the instructions are executed one after another, with no concurrent operations. The model contains common instructions: arithmetic (add, subtract, multiply, divide, remainder, bitwise shifts), data movement (load the content of a memory cell, store a number in a memory cell, copy the content of a memory cell to another), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time. Note that comparing two numbers is also assumed to take a constant amount of time and it is also a common assumption that basic operations on arrays (like accessing or updating the values found at a certain position of the array) containing a polynomial (in n) number of $\mathcal{O}(\log n)$-bit integer elements, are carried out in constant time. Basically, this model allows us to measure the number of instructions executed in an algorithm, making abstraction of the time spent to execute each of the basic instructions.

We hope that these remarks will make the reader note the difference between the model used in Theorem 4 and that used later, and understand the difference between the theoretical flavor of the previous result and the more applicative flavor of the results that follow.

4. Combinatorial and algorithmic prerequisites

Besides the assumptions on the computational model we use, discussed in the last section, a few other assumptions are made in the following.

In the upcoming algorithmic problems, whenever we are given as input a word w of length n we assume that the symbols of w are in fact integers from $\{1, \ldots, n\}$ (i.e., $alph(w) \subseteq \{1, \ldots, n\}$), and w is seen as a sequence of integers. This is a common assumption in algorithmic on words (see, e.g., the discussion in [9]). Also, in the following we assume that all logarithms are in base 2.

In all our algorithms we compute different functions

$$f : \{1, \ldots, n\}^k \times \{1, \ldots, m\}^\ell \to S,$$

where n is the size of the input, $m \leqslant n$, k and ℓ are constants, and S is a set whose elements can be stored in a constant number of memory words. Such a function f is canonically implemented as a $k+\ell$-dimensional array H_f, where $H_f[i_1]\ldots[i_{k+\ell}] = f(i_1, \ldots, i_{k+\ell})$. In such an implementation, and using the RAM with logarithmic word size model, the space needed to store such a function is $\mathcal{O}(n^k m^\ell)$. Without the danger of any confusion, we will work directly with the functions as arrays, keeping in mind this implementation.

Combinatorics on words The following well known results (see [5]) are useful to our algorithms. The first result regards the lengths of the primitively rooted squares occurring as prefixes of a given word.

Lemma 1. *Let u_1, u_2, u_3 be primitive words, such that $|u_1| < |u_2| < |u_3|$ and u_i^2 is a prefix of a word v, for all $1 \leqslant i \leqslant 3$. Then $2|u_1| < |u_3|$.*

By the previous lemma, the number of primitively rooted squares occurring as prefixes of a given word can be bounded.

Corollary 2. *For a word v with $|v| = n$, we have that*

$$\left|\{u \mid u \text{ primitive}, u^2 \text{ is a prefix of } v\}\right| \leqslant 2\log n.$$

Identical results can be derived for the primitively rooted squares occurring as suffixes of a given word.

Data structures For a string u of length n, over an alphabet $V \subseteq \{1, \ldots, n\}$, we define a suffix-array data structure that contains two arrays Suf_u, which is a permutation of $\{1, \ldots, n\}$, and LCP_u with n elements from $\{0, 1, \ldots, n-1\}$. Basically, Suf_u is defined such that $u[Suf_u[i]..n]$ is the ith suffix of u, in the lexicographical order. The array LCP_u is defined by $LCP_u[1] = 1$ and $LCP_u[r]$ is the length of the longest common prefix of $u[Suf_u[r-1]..n]$ and $u[Suf_u[r]..n]$. These data structures are constructed in time $\mathcal{O}(n)$. For more details, see [9], and the references therein. Moreover, one can process the array LCP_u in linear time $\mathcal{O}(n)$ in order to return in constant time the answer to queries "What is the length of the longest common prefix of $u[i..n]$ and $u[j..n]$?", denoted $LCPref_u(i, j)$. The idea is to first compute a structure S_u that associates to each i the value $S_u[i] = \ell$ if and only if $i = Suf_u[\ell]$; in other words S_u is the inverse permutation of Suf_u. Further we compute in linear time a range minimum query data structure for the array LCP_u (see [8]), and return in constant time the answer to queries "What is the minimum number from $LCP_u[i], LCP_u[i+1], \ldots, LCP_u[j]$?". Now, $LCPref_u(i, j)$ is obtained as the minimum from $LCP_u[i'+1], \ldots, LCP_u[j']$, where $i' = \min(S_u[i], S_u[j])$ and $j' = \max(S_u[i], S_u[j])$.

Lemma 2. *Let $w \in V^*$ be a word of length n. We can compute the values $per(i)$, the period of $w[1..i]$, for all $i \in \{1, \ldots, n\}$ in linear time $\mathcal{O}(n)$.*

Proof. One may note that the result follows from the preprocessing part of the classical Knuth–Morris–Pratt algorithm. Alternatively, a proof based on *LCPref* queries can be easily given.

Note that $per(1) = 1$ and $per(i) \leqslant per(j)$ for $i < j$. Consequently, to compute $per(i+1)$ we compute the minimum $j \geqslant per(i)$ such that $LCPref_w(w[1..i+1], w[j..i+1]) = i - j + 2$. Using this idea, $per(i+1)$ is computed in $per(i+1) - per(i)$ time. Thus, computing all the values $per(i)$ for $i \in \{1, \ldots, n\}$ takes $\mathcal{O}(\sum_{i=2}^{n}(per(i) - per(i-1)))$ time to which the time needed to construct data structures that allow us answer *LCPref* for w is added. The statement of our lemma follows. □

The following lemma is a simple consequence of Lemma 2.

Lemma 3. *Let $w \in V^*$ be a word of length n. We can compute the values $per(i, j)$, the period of $w[i..j]$, for all $i, j \in \{1, \ldots, n\}$ in quadratic time $\mathcal{O}(n^2)$.*

Note that we can store all the values $per(i, j)$ in $\mathcal{O}(n^2)$ space, as we have explained in the beginning of this section.

Assume, in the following, that $w \in V^*$ is a word of length n. For each $i \leqslant n$ we define the sets P_i and S_i as follows:

$$P_i = \{|u| \mid u \text{ is a primitive word such that } u^2 \text{ is a prefix of } w[i..n]\},$$

$$S_i = \{|u| \mid u \text{ is a primitive word such that } u^2 \text{ is a suffix of } w[1..i]\}$$

Moreover, by combining the results of Lemma 3 and Corollary 2, we get the following. Let us now define

Lemma 4. *Let $w \in V^*$ be a word of length n. We can compute in quadratic time and $\mathcal{O}(n \log n)$ space the sets P_i and S_i, for all $i \in \{1, \ldots, n\}$.*

Proof. We give the argument only for the sets P_i, as showing the statement of the lemma for the sets S_i is just analogous.

Let us note first that we can decide whether $w[i..j]$ is a primitively rooted square just by checking whether $|j - i + 1| = 2per(i, j)$; this takes $\mathcal{O}(1)$ time. Furthermore, we store for each $i \in \{1, \ldots, n\}$ the set P_i that contains all the numbers ℓ such that $u[i..i+2\ell-1]$ is a primitively rooted square. Each such set can be stored as an array with at most $2 \log n$ elements, each such element being a number less than n. Thus, in our model, P_i is stored in $\mathcal{O}(\log n)$ memory words. Moreover, the time needed to compute this array is determined by the fact that we go through all numbers $\ell \in \{1, \ldots, \lfloor \frac{n-i+1}{2} \rfloor\}$; consequently, this time is linear in n.

According to the above, the total time needed to compute the sets P_i for all $1 \leqslant i \leqslant n$ is $\mathcal{O}(n^2)$ and the space needed to store them is $\mathcal{O}(n \log n)$. □

Finally, it is easy to see that given a prefix $w[i..j]$ of $w[i..n]$ we can compute the maximum power ℓ such that $w[i..j]^\ell$ is also a prefix of $w[i..n]$ in constant time. In fact, $\ell = \lfloor \frac{LCPref(w[i..n], w[j+1..n])}{j-i+1} \rfloor + 1$.

All the notions introduced in this section will become useful in the following two sections.

5. Complexity of the prefix–suffix duplication languages

Recall that the problem we try to solve is that of deciding, for two given words x and w, whether w can be obtained by iteratively applying the prefix–suffix duplication to x.

A preprocessing step in our approach is to compute for the word w, as previously described, data structures allowing us to answer in constant time *LCPref* queries, as well as the data structures from Lemmas 2, 3, and 4.

Further, the main idea of our algorithm is to compute, by dynamic programming, the function $\Lambda(i, j)$ defined by:

$$\Lambda(i, j) = \begin{cases} 1, & \text{if } w[i..j] \in PSD^*(x), \\ 0, & \text{otherwise} \end{cases}$$

In computing efficiently the values of the function $\Lambda(\cdot, \cdot)$ we use the following result.

Lemma 5. *Let w be a word of length n. The word $w[i..j]$ can be obtained by prefix–suffix duplications from x if and only if one of the following condition holds*

1. *$w[i..j] = x$,*
2. *there exist a word $v \in P_i$ such that $w[i + |v|..j]$ can be obtained by prefix–suffix duplications from x,*
3. *there exist a word $v \in S_i$ such that $w[i..j - |v|]$ can be obtained by prefix–suffix duplications from x.*

Proof. Assume that $w[i..j]$ can be obtained by prefix–suffix duplications from x. If $w[i..j] \neq x$ then there exists a word u such that $w[i..j] = uuw[i'..j]$ and $uw[i'..j]$ can be obtained by prefix–suffix duplications from x, or there exists a word u such that $w[i..j] = w[i..j']uu$ and $w[i..j']u$ can be obtained by prefix–suffix duplications from x. We only consider the first case, as the second one is similar.

Algorithm 3 Compute the function Λ. **Input**: words x and w.

```
1: Compute LCPref data structures for the word wx.
2: Compute for w the sets P_i and S_i, for all i ∈ {1, ..., n}.
3: Set all the values Λ(i, j) to 0.
4: for ℓ = 1 to n do
5:    for i = 1 to n + ℓ − 1 do
6:       Set j = i + ℓ − 1.
7:       if w[i..j] = x then Set Λ(i, j) = 1; Go to line 13.
8:       for v in P_i such that 2|v| ⩽ j − i + 1 do
9:          if Λ(i + |v|, j) = 1 then Set Λ(i, j) = 1; Go to line 13.
10:      end for
11:      for v in S_i do
12:         if Λ(i, j − |v|) = 1 then Set Λ(i, j) = 1; Go to line 13.
13:      end for
14:   end for
15: end for
```

If u is primitive, we get the conclusion directly. So, let us assume that $u = v^k$, for some word v and $k > 1$. We know that $v^k w[i'..j]$ can be obtained by prefix–suffix duplications from x, thus, $v^{k+\ell} w[i'..j]$ can be obtained by prefix–suffix duplications from x, for all $\ell > 0$. Hence, $w[i..j] = uuw[i'..j] = v^{2k} w[i'..j]$ can be obtained by prefix–suffix duplications from x. □

In order to compute $\Lambda(i, j)$ we use the strategy described in Algorithm 3.

The complexity of Algorithm 3 is clearly $\mathcal{O}(n^2 \log n)$, due to the **for**-instructions in lines 3 and 4, that range over sets with n elements, and, respectively, in lines 7 and 10, that range over sets with at most $2 \log n$ elements. As far as the other instructions are concerned, we note that the instructions 1 and 2 can be implemented in linear time (by the arguments given in the previous section), while all the others can be clearly performed in constant time in our computational model. The only case that requires special attention is that of the test performed in instruction 6: we can test whether $w[i..j] = x$ or not by checking whether $LCPref_{wx}(i, n+1) = |x|$ and $j - i + 1 = |x|$ or, respectively, not. From Lemma 5 it follows that this algorithm computes the values of the function $\Lambda(\cdot, \cdot)$ correctly. Clearly, w can be obtained by prefix–suffix duplications from x if and only if $\Lambda(1, n) = 1$.

By Lemmas 3 and 4 we get that the space complexity of this algorithm is quadratic. Therefore, we showed the following result.

Theorem 5. *The membership problem for prefix–suffix duplication languages can be solved in $\mathcal{O}(n^2 \log n)$ time and $\mathcal{O}(n^2)$ space, where n is the length of the input word.*

6. Prefix–suffix duplication distance

In this section we present two polynomial time algorithms for computing the prefix–suffix duplication distance. The first one has cubic time complexity and quadratic space complexity while the second one runs faster, namely it runs in $\mathcal{O}(n^2 \log n)$ time, but, compared to the aforementioned algorithm, as well as to the membership algorithm from the previous section, it consumes more space, namely $\mathcal{O}(n^2 \log n)$.

6.1. A dynamic programming algorithm

Let x, w be the input words over an alphabet V. Without loss of generality we may assume that $m = |x| \leqslant |w| = n$ (otherwise, we interchange the words). Recall that we are interested in computing $\pi(x, w)$, the prefix–suffix duplication distance between these two words. In our case, this distance is the minimum number of prefix–suffix duplication steps needed to obtain w from x, or ∞ if $w \notin PSD^*(x)$.

We first define two functions, that are used in our approach:

$$P, S : \{1, \ldots, n\}^2 \to 2^{\{1,\ldots,n\}}$$

$$P(i, j) = \left\{ p \;\middle|\; {}_{w[i..j]}(i, p) \text{ is a duplication in } w[i..j] \right\},$$

$$S(i, j) = \left\{ p \;\middle|\; (j, p)_{w[i..j]} \text{ is a duplication in } w[i..j] \right\}.$$

Informally, $P(i, j)$ is the set of the lengths of all prefixes of $w[i..j]$ which are duplications, while $S(i, j)$ is the set of the lengths of all suffixes of $w[i..j]$ which are duplications. It is straightforward that the two functions can be computed in $\mathcal{O}(n^3)$. A bit more complicated is to note that these sets can be stored in $\mathcal{O}(n^2)$ space. The key observation is that $P(i, j) \subseteq P(i, j+1)$ and $S(i+1, j) \subseteq S(i, j)$. Moreover, $|P(i, j) \setminus P(i, j+1)| \leqslant 1$ and $|S(i, j) \setminus S(i+1, j)| \leqslant 1$. Thus, to store the set $P(i, j+1)$ we do not have to store all its elements together, but rather the only element (if any) that appears in this set and did not appear in $P(i, j)$; a similar approach can be used to store the sets $S(i, j)$, for all i and j.

Algorithm 4 Procedure for computing σ. **Input:** words w and x, $|w| \geqslant |x|$.

```
Compute LCPref data structures for the word wx;
Compute the sets P and S for the word w;
for all 1 ⩽ i ⩽ n do
    for all 1 ⩽ j ⩽ n − k + 1 do
        if j − i + 1 = |x| and w[i..j] = x then σ(i, j) = 0;
        if j − i + 1 = |x| and w[i..j] ≠ x then σ(i, j) = ∞;
        if j − i + 1 < |x| then σ(i, j) = ∞;
    end for
end for
for all m + 1 ⩽ k ⩽ n do
    for all 1 ⩽ i ⩽ n − k + 1 do
        m1 = min_{p∈S(i,i+k−1)}{σ(i, i + k − 1 − p)};
        m2 = min_{p∈P(i,i+k−1)}{σ(i + p, i + k − 1)};
        σ(i, i + k − 1) = min(m1, m2) + 1;
    end for
end for
```

More precisely, in order to achieve the announced space complexity bound we have to store, instead of the functions P and S, just two auxiliary functions $P', S' : \{1, \ldots, n\}^2 \to 2^{\{1,\ldots,n\}}$ defined by:

$$P'(i,i) = P(i,i), \qquad P'(i,j) = P(i,j) \setminus P(i,j-1) \quad \text{for } j > i, \qquad P'(i,j) = \emptyset \quad \text{for } j < i,$$

$$S'(j,j) = S(j,j), \qquad S'(i,j) = S(i,j) \setminus S(i+1,j) \quad \text{for } j > i, \qquad S'(i,j) = \emptyset \quad \text{for } j < i.$$

Note that all the sets $P'(i,j)$ and $S'(i,j)$ are, in fact, either singletons or empty sets. Clearly, $P'(i,j) \cap P'(i,j') = \emptyset$ for $j \neq j'$. Storing these new functions requires only $\mathcal{O}(n^2)$ space as for every $i \in \{1, \ldots, n\}$ we have that each $\ell \in \{1, \ldots, n\}$ belongs to at most one of the sets $P'(i,j)$, for $j \geqslant i$. We can easily compute them in $\mathcal{O}(n^3)$ time. Now, once the function P' computed and stored, we can access the elements of $P(i,j)$ just by accessing one by one the elements of $P'(i,i)$, $P'(i,i+1), \ldots, P'(i,j)$. Similarly, once the function S' was computed and stored, we can access the elements of $S(i,j)$ just by accessing one by one the elements of $S'(i,j), S'(i+1,j), \ldots, S'(j,j)$. It is straightforward that using this implementation we can go through the elements of $P(i,j)$ and $S(i,j)$ in $\mathcal{O}(n)$ time.

Following the basic remark, used also in Algorithm 3, that in the process of obtaining w starting from x by iteratively applying prefix–suffix duplications, all intermediate words are subwords of w, we will use again a dynamic programming approach.

We now define the function

$$\sigma : \{1, \ldots, n\}^2 \to \{0, 1, \ldots, n\} \cup \{\infty\},$$

$$\sigma(i,j) = \pi\big(x, w[i..j]\big), \quad \text{if } |j - i + 1| \geqslant |x|,$$

$$\sigma(i,j) = \infty, \quad \text{otherwise.}$$

Clearly, $\sigma(1,n) = \pi(x,w)$. The values $\sigma(i,j)$ can be computed in the increasing order of $j - i$ as follows:

$$\sigma(i,j) = \begin{cases} \infty, & \text{if } j - i + 1 < |x|, \\ 0, & \text{if } w[i..j] = x, \\ \infty, & \text{if } j - i + 1 = |x| \text{ and } w[i..j] \neq x, \\ \min(\min_{p \in S(i,j)}\{\sigma(i, j-p)\}, \min_{p \in P(i,j)}\{\sigma(i+p, j)\}) + 1, & \text{if } j - i + 1 > |x|. \end{cases}$$

Let us determine the time complexity of our algorithm that computes the distance between x and w, according to the strategy described above. The preprocessing phase, i.e., computing the sets $P(i,j)$ and $S(i,j)$ for all i and j, takes $\mathcal{O}(n^3)$ time. We now determine the time complexity of the procedure presented in Algorithm 4, in which the values $\sigma(i,j)$ are computed. Note that Algorithm 4 initializes first all values $\sigma(i,j)$, for all $i \leqslant j$, and $\sigma(i,j)$, with $|j - i + 1| = |x|$ and $w[i..j] \neq x$, to ∞, while all values $\sigma(i,j)$, with $w[i..j] \neq x$, are initialized to 0. These initializations can be done in $\mathcal{O}(n^2)$ time, provided that we check whether $w[i..j] = x$ using $LCPref_{wx}$ queries, just like in Algorithm 3. Going through every set $P(i,j)$ and $S(i,j)$ requires $\mathcal{O}(n)$ time (using the implementation described above), so computing each value $\sigma(i,j)$, with $j \geqslant i$, is done in $\mathcal{O}(n)$ time. Therefore, the total running time of the procedure is clearly upper bounded by $\mathcal{O}(n^3)$.

The space used by the above algorithm is upper bounded by the space needed to store the functions P and S, which is quadratic, and the space needed to store the function σ, also quadratic.

In conclusion, we have the next result:

Theorem 6. *Let x, w be two words such that $|x| \leqslant |w| = n$. The prefix–suffix duplication distance between x and w can be computed in $\mathcal{O}(n^3)$ time and $\mathcal{O}(n^2)$ space.*

6.2. A faster algorithm

We now present our second algorithm. It is worth mentioning that this algorithm puts together ideas from the efficient solution of the membership problem (that is, Algorithm 3) and the dynamic programming approach of the previous section, in order to compute faster the prefix suffix duplication distance between two words.

We work under the same assumptions as in the previous section: we are given two words w and x, with $|w| = n \geqslant |x| = m$.

As in the all the algorithms described so far, we construct data structures needed to answer in constant time *LCPref* queries for the words w and wx, as well as the sets P_i and S_i for the word w.

Let us assume that for all $i \in \{i, \ldots, n\}$ the elements of the sets P_i and S_i are ordered increasingly (actually, this requirement can be easily fulfilled, even from their initial construction). Accordingly, these sets are stored as (ordered) arrays, such that the element $P_i[k]$ (respectively, $S_i[k]$) stored on position k in the array corresponding to P_i (respectively, S_i) stores the kth element of the ordered set P_i (respectively, S_i), i.e., the length of the kth shortest primitively rooted square occurring at position i (respectively, ending at position i). Note that if $P_i[k] = \ell$ and $w[i..i+\ell-1]^2$ is a prefix of $w[j..n]$, then $P_j[k] = \ell$; indeed, a word x^2 with x primitive and $|x| < \ell$ is a prefix of $w[i..n]$ if and only if it is a prefix of $w[i..i+\ell-1]^2$ if and only if it is a prefix of $w[j..n]$. Similarly, if $S_i[k] = \ell$ and $w[i-\ell+1..i]^2$ is a suffix of $w[1..j]$, then $S_j[k] = \ell$.

Recall that each of these arrays has at most $2\log n$ elements. For simplicity, we assume that, in our implementation, each such array has exactly $2\log n$ elements, and if the set of primitively rooted squares occurring at position i in w has ℓ elements with $\ell < 2\log n$ elements, then only the first ℓ elements of the ordered array storing P_i are defined.

To compute the prefix–suffix duplication distance between x and w, we design several data structure, more complex than the ones used in the previous sections.

Namely, we first define the function

$$\Pi_p : \{1, \ldots, n\}^2 \times \{1, \ldots, 2\log n\} \to \big(\{1, \ldots, n\} \cup \{\infty\}\big) \times \{1, \ldots, 2\log n\}.$$

We have $\Pi_p(i, j, k) = (s, i')$, with $s \neq \infty$, if and only if there exists $s \geqslant n$ such that s is the minimum number of prefix or suffix duplication steps needed to obtain $w[i..j]$ from x, such that the last duplication step in this succession of duplication steps was the addition of a factor v^ℓ as a prefix to a word $w[i'..j]$, where $v = w[i..i + P_i[k] - 1]$ and $\ell \geqslant 1$; moreover, we require that i' is minimum with the above property. We have $\Pi_p(i, j, k) = (\infty, 1)$ whenever $w[i..j]$ cannot be obtained from x in the way described above.

We define and work, in a similar fashion, with the function

$$\Pi_s : \{1, \ldots, n\}^2 \times \{1, \ldots, 2\log n\} \to \big(\{1, \ldots, n\} \cup \{\infty\}\big) \times \{1, \ldots, 2\log n\}.$$

We have $\Pi_s(i, j, k) = (s', j')$, with $s' \neq \infty$, if and only if there exists $s' \leqslant n$ such that s' is the minimum number of prefix or suffix duplication steps needed to obtain $w[i..j]$ from x, such that the last duplication step of this derivation was the addition of a factor v^ℓ as a suffix to a word $w[i..j']$, with $v = w[j - S_j[k] + 1..j]$ and $\ell \geqslant 1$; moreover, we ask that j' is maximum with the above property. We have $\Pi_s(i, j, k) = (\infty, 1)$ whenever $w[i..j]$ cannot be obtained from x in the way described above.

As in the previous section, we now define the function:

$$\sigma : \{1, \ldots, n\}^2 \to \{0, 1, \ldots, n\} \cup \{\infty\},$$
$$\sigma(i, j) = \pi\big(x, w[i..j]\big), \quad \text{if } |j - i + 1| \geqslant |x|,$$
$$\sigma(i, j) = \infty, \quad \text{otherwise.}$$

Clearly,

$$\sigma(i, j) = \min\Big(\big\{\Pi_p(i, j, k) \;\big|\; k \leqslant 2\log n\big\} \cup \big\{\Pi_s(i, j, k) \;\big|\; k \leqslant 2\log n\big\}\Big).$$

Further, we show how the values of the function $\Pi_p(\cdot, \cdot, \cdot)$ can be computed.

Let us note the next straightforward remarks:

Remark 1. Let $w[i..j]$ be a factor of w with $j - i + 1 > |x|$ and let v be a primitive prefix of $w[i..j]$ such that $P_i[k] = |v|$. Also, let ℓ be maximum such that v^ℓ is a prefix of $w[i..j]$ and assume $\ell \geqslant 3$, i.e., v^2 is also a prefix of $w[i + |v|..n]$; in this case, $P_{i+|v|}[k] = |v|$. Finally, assume that $\Pi_p(i + |v|, j, k) = (s, i')$ and $\sigma(i + |v|, j) = s$; it is not hard to see that $i' \leqslant i + |v|\lfloor\frac{\ell+1}{2}\rfloor$, and the equality holds only if ℓ is odd. We have the following:

- If $i' \leqslant i + |v|\lfloor\frac{\ell}{2}\rfloor$, then $w[i..j]$ can be obtained by prefix suffix duplication, such that a power of v is appended in the last step, in s steps (for instance, v^t can be added to $w[i'..j]$, where $t = \frac{i'-i}{|v|}$). Note that $i' - i$ is always divisible by $|v|$.
- If $i' = i + |v|\lfloor\frac{\ell+1}{2}\rfloor > i + |v|\lfloor\frac{\ell}{2}\rfloor$ (hence, ℓ is odd), then $w[i..j]$ can be obtained by prefix suffix duplication, such that a power of v is appended in the last step, in $s + 1$ steps (more precisely, v is added to $w[i + |v|..j]$) and cannot be obtained by prefix suffix duplication, such that a power of v is appended in the last step, in s steps or less. □

A similar remark can be derived for Π_s.

The following two remarks state the most important step of our approach. The first remark shows the basic step in a dynamic programming approach used to compute the values of the Π_p and Π_s.

Remark 2. Let $w[i..j]$ be a factor of w with $j-i+1>|x|$ and let v be a primitive prefix of $w[i..j]$ such that $P_i[k]=|v|$ and $\ell=2$ is maximum such that v^ℓ is a prefix of $w[i..j]$. Then $\Pi_p(i,j,k)=\sigma(i+|v|,j)+1$.

One can obtain an analogous result for Π_s.

The second remark shows how we can compute the rest of the values of these functions by induction, and it formalizes the core of our dynamic programming.

Remark 3. Let $w[i..j]$ be a factor of w with $j-i+1>|x|$ and let v be a primitive prefix of $w[i..j]$ such that $P_i[k]=|v|$. Also, let ℓ be maximum such that v^ℓ is a prefix of $w[i..j]$ and assume $\ell\geqslant 3$, i.e., v^2 is also a prefix of $w[i+|v|..n]$; in this case, $P_{i+|v|}[k]=|v|$. Then $\Pi_p(i,j,k)$ is the minimum of the values m_1, m_2, and m_3 where:

- $m_1=\sigma(i+|v|,j)+1$,
- $m_2=s+1$, if $\Pi_p(i+|v|,j,k)=(s,i')$, $i'>i+|v|\lfloor\frac{\ell}{2}\rfloor$, and $\sigma(i+|v|,j)=s$, or $m_2=\infty$, otherwise,
- $m_3=s$, if $\Pi_p(i+|v|,j,k)=(s,i')$, $i'\leqslant i+|v|\lfloor\frac{\ell}{2}\rfloor$, and $\sigma(i+|v|,j)=s$, or $m_3=\infty$, otherwise. □

Again, a similar remark can be derived for the case when we are interested in the way the values of Π_s are computed.

Further, it is clear that the values of $\sigma(\cdot,\cdot)$ can be computed by a dynamic programming algorithm.

- If $w[i..j]=x$ we set $\sigma(i,j)=0$, as well as $\Pi_p(i,j,k)=0$ and $\Pi_s(i,j,t)=0$ for all k and t as in the definition of Π_p and Π_s.
- If $|w[i..j]|>x$ we compute $\Pi_p(i,j,k)$ and $\Pi_s(i,j,t)$, for all k and t as in the definition of Π_p and Π_s, as described above. We set $\sigma(i,j)$ to be the minimum of these values.

Now, this strategy can be plugged into an algorithm similar to Algorithms 3 and 4, and this solves completely our problem. More precisely, we compute the values of the function σ and return $\sigma(1,n)$ as the distance $\pi(x,w)$. We only have to determine the complexity of this implementation. To do that, we note that, for some i, j, and k (respectively, i, j, and t), the values $\Pi_p(i,j,k)$ (or $\Pi_s(i,j,t)$) are computed in constant time using the previously defined data structures. Moreover, for some fixed i and j, the number of values $\Pi_p(i,j,k)$ and $\Pi_s(i,j,t)$ we have to compute is $\mathcal{O}(\log n)$. Once these values computed, we get $\sigma(i,j)$ in $\mathcal{O}(\log n)$ time. Therefore, the overall complexity of our algorithm is $\mathcal{O}(n^2\log n)$.

The space complexity is determined by the space needed to store the arrays Π_p and Π_s. Accordingly, it is $\mathcal{O}(n^2\log n)$.

In conclusion, the following theorem follows from the remarks presented above.

Theorem 7. *Let x, w be two words such that $|x|\leqslant|w|=n$. The prefix–suffix duplication distance between x and w can be computed in $\mathcal{O}(n^2\log n)$ time and $\mathcal{O}(n^2\log n)$ space.*

7. Final remarks

It remains open whether the algorithms presented here can be improved or not. In our view, a similar investigation on the bounded prefix–suffix duplications, namely duplications in which the length of the prefix or suffix that is to be duplicated is bounded by a constant, is worth pursuing. In fact, such a situation seems closer to our biological motivation, as telomeres are tandem repeats of just a small number of nucleotides, so, the length of the duplicated prefixes/suffixes should also be small. Some of the questions addressed in the section devoted to prefix–suffix duplication languages appear to have easier answers. Probably, this restriction leads to more efficient duplication distance algorithms, as well.

It is our hope that the prefix–suffix duplication distance will prove useful in future studies of the duplication architecture of chromosomes, studies that are now possible due to the genome sequencing projects.

Acknowledgments

The work of Florin Manea is supported by the *DFG* grant 596676. We gratefully acknowledge the reviewers' comments and suggestions which improved the presentation.

References

[1] G. Benson, L. Dong, Reconstructing the duplication history of a tandem repeat, in: T. Lengauer, R. Schneider, P. Bork, D.L. Brutlag, J.I. Glasgow, H.-W. Mewes, R. Zimmer (Eds.), Proc. Int. Conf. Intell. Syst. Mol. Biol., AAAI, Heidelberg, Germany, 1999, pp. 44–53.
[2] D.A. Benson, Tandem repeat finder: A program to analyze DNA sequences, Nucleic Acids Res. 27 (2) (1999) 573–580.
[3] D.P. Bovet, S. Varricchio, On the regularity of languages on a binary alphabet generated by copying systems, Inf. Process. Lett. 44 (3) (1992) 119–123.

[4] S.R. Chan, E.H. Blackburn, Telomeres and telomerase, Philos. Trans. R. Soc. Lond. B, Biol. Sci. 359 (2004) 109–121.
[5] M. Crochemore, W. Rytter, Squares, cubes, and time-space efficient string searching, Algorithmica 13 (5) (1995) 405–425.
[6] J. Dassow, V. Mitrana, Gh. Păun, On the regularity of duplication closure, Bull. Eur. Assoc. Theor. Comput. Sci. 68 (1999) 133–136.
[7] A. Ehrenfeucht, G. Rozenberg, On the separating power of EOL systems, RAIRO Inform. Théor. 17 (1) (1983) 13–22.
[8] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, NY, USA, 1997.
[9] J. Kärkkäinen, P. Sanders, S. Burkhardt, Linear work suffix array construction, J. ACM 53 (2006) 918–936.
[10] P. Leupold, V. Mitrana, J.M. Sempere, Formal languages arising from gene repeated duplications, in: N. Jonoska, G. Paun, G. Rozenberg (Eds.), Aspect of Molecular Computing, in: LNCS, vol. 2950, Springer-Verlag, Berlin, 2004, pp. 301–310.
[11] P. Leupold, Reducing repetitions, in: J. Holub, J. Zdarek (Eds.), Prague Stringology Conference, 2009, pp. 225–236.
[12] J.P. Murnane, Telomere dysfunction and chromosome instability, Mutat. Res. 730 (2012) 28–36.
[13] W. Ogden, A helpful result for proving inherent ambiguity, Math. Syst. Theory 2 (3) (1968) 191–194.
[14] C. Papadimitriou, Computational Complexity, Addison–Wesley, 1994.
[15] E. Pennisi, Genome duplications: The stuff of evolution? Science 294 (2001) 2458–2460.
[16] R.J. Preston, Telomeres, telomerase and chromosome stability, Radiat. Res. 147 (1997) 529–534.
[17] G. Rozenberg, A. Salomaa, Handbook of Formal Languages, vols. I–III, Springer-Verlag, Berlin, 1997.
[18] D.B. Searls, The computational linguistics of biological sequences, in: L. Hunter (Ed.), Artificial Intelligence and Molecular Biology, MIT Press, Cambridge, MA, 1993, pp. 47–120.
[19] D. Sokol, G. Benson, J. Tojeira, Tandem repeats over the edit distance, Bioinformatics 23 (2) (2007) 30–35.
[20] L.D. Stein, End of the beginning, Nature 431 (2004) 915–916.
[21] M.-w. Wang, On the irregularity of the duplication closure, Bull. Eur. Assoc. Theor. Comput. Sci. 70 (2000) 162–163.
[22] I. Wapinski, A. Pfeffer, N. Friedman, A. Regev, Natural history and evolutionary principles of duplication in fungi, Nature 449 (2007) 54–61.